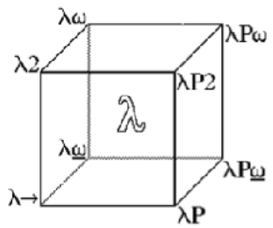
$$\left(\begin{array}{c}\textit{Curry}\\ \textit{Howard}\end{array}\right.$$

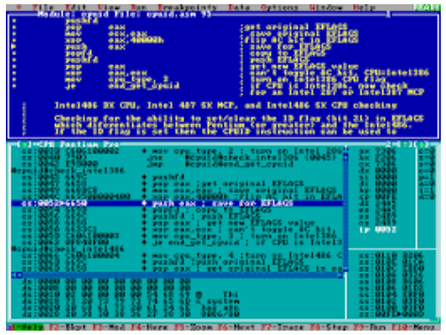


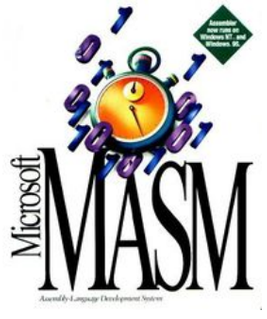
$$\left.\textit{FP FRP}\right) \in \Gamma$$

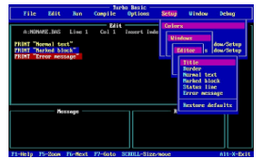
$$\Gamma \vdash \textit{Web applications}$$

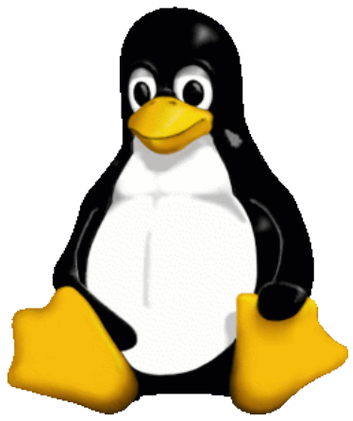
Eugene Naumenko

eugene@traversable.one









SQL

XSLT

PROLOG

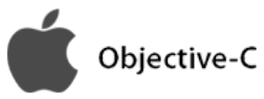

JS


php


[ ]


ERLANG


Eiffel


Objective-C

FP








Agda



OCaml

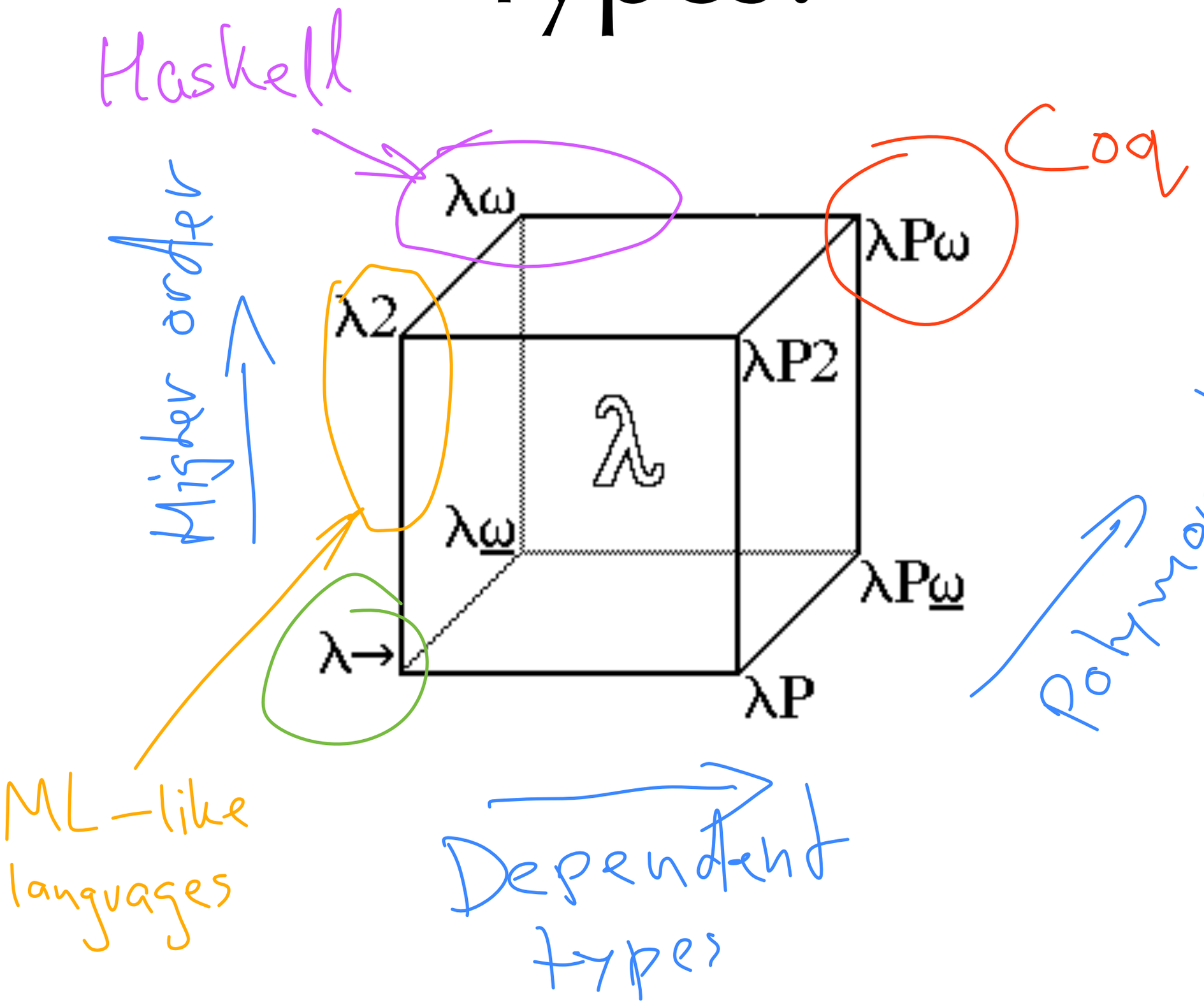
Idris


NOT DEFINITIVE →

How Functional Is My Language?	Agda	C	C++11	C++17	Clojure	Common LISP	F#	Haskell	Java 8	JavaScript	Perl	Python	Racket	Rust	Scala	Scheme	SML & OCaml
pure	Y	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N	N
proper tail calls	N/A	N	N	N	M	M	Y	Y	N	N	N	N	Y	N	M	Y	Y
higher-order	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
function type	Y	M	Y	Y	N/A	N/A	Y	Y	M	N/A	N/A	N/A	N/A	Y	Y	N/A	Y
lambda	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
statement lambda	N/A	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y
full block scope	Y	N	Y	Y	Y	Y	Y	Y	M	Y	Y	M	Y	Y	Y	Y	Y
downward funargs	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
upward funargs	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
algebraic datatypes	Y	N	N	Y	M	N	Y	Y	N	N	N	N	Y	Y	Y	N	Y
pattern matching	Y	N	N	N	M	M	Y	Y	N	N	N	N	Y	Y	Y	M	Y
expression *if*	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
expression *case*	Y	N	N	M	Y	Y	Y	Y	N	N	Y	N	Y	Y	Y	Y	Y
expression block/let	Y	N	M	M	Y	Y	Y	Y	M	M	Y	N	Y	Y	Y	Y	Y

Y	Yes, supported
M	Meh, somewhat supported
N	No, not supported
N/A	Does not apply

Types!


Haskell
Coq
λω
λ2
λP2
λPω
λ
λω
λP2ω
λ→
λP
λP
Higher order
Polymorphism
ML-like languages
Dependent types

Curry-Howard

propositions *as* types

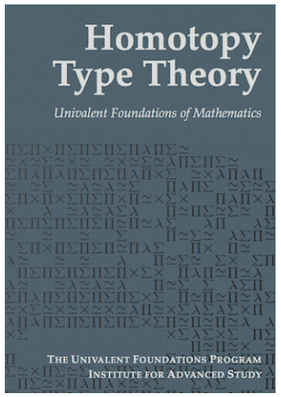
proofs *as* programs

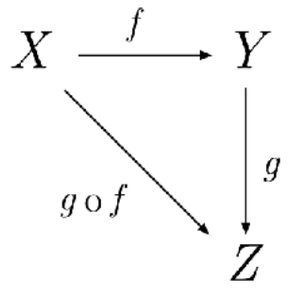
normalisation of proofs *as* evaluation of programs

> *The Curry–Howard isomorphism is a **profound insight** about our **universe**. It allows us to analyze **mathematical theorems** through the lens of **functional programming**. What's better is that often even "boring" mathematical theorems are interesting when expressed as types.*

Curry-Howard-Lambek & more

Natural Deduction	↔	Typed Lambda Calculus
Gentzen (1935)		Church (1940)
Type Schemes	↔	ML Type System
Hindley (1969)		Milner (1975)
System F	↔	Polymorphic Lambda Calculus
Girard (1972)		Reynolds (1974)
Modal Logic	↔	Monads (state, exceptions)
Lewis (1910)		Kleisli (1965), Moggi (1987)
Classical-Intuitionistic Embedding	↔	Continuation Passing Style
Gödel (1933)		Reynolds (1972)
Linear Logic	↔	Session Types
Girard (1987)		Honda (1993)






Cubical Type Theory: a constructive interpretation of the univalence axiom

THE CURRY-HOWARD-LAMBEK CORRESPONDENCE

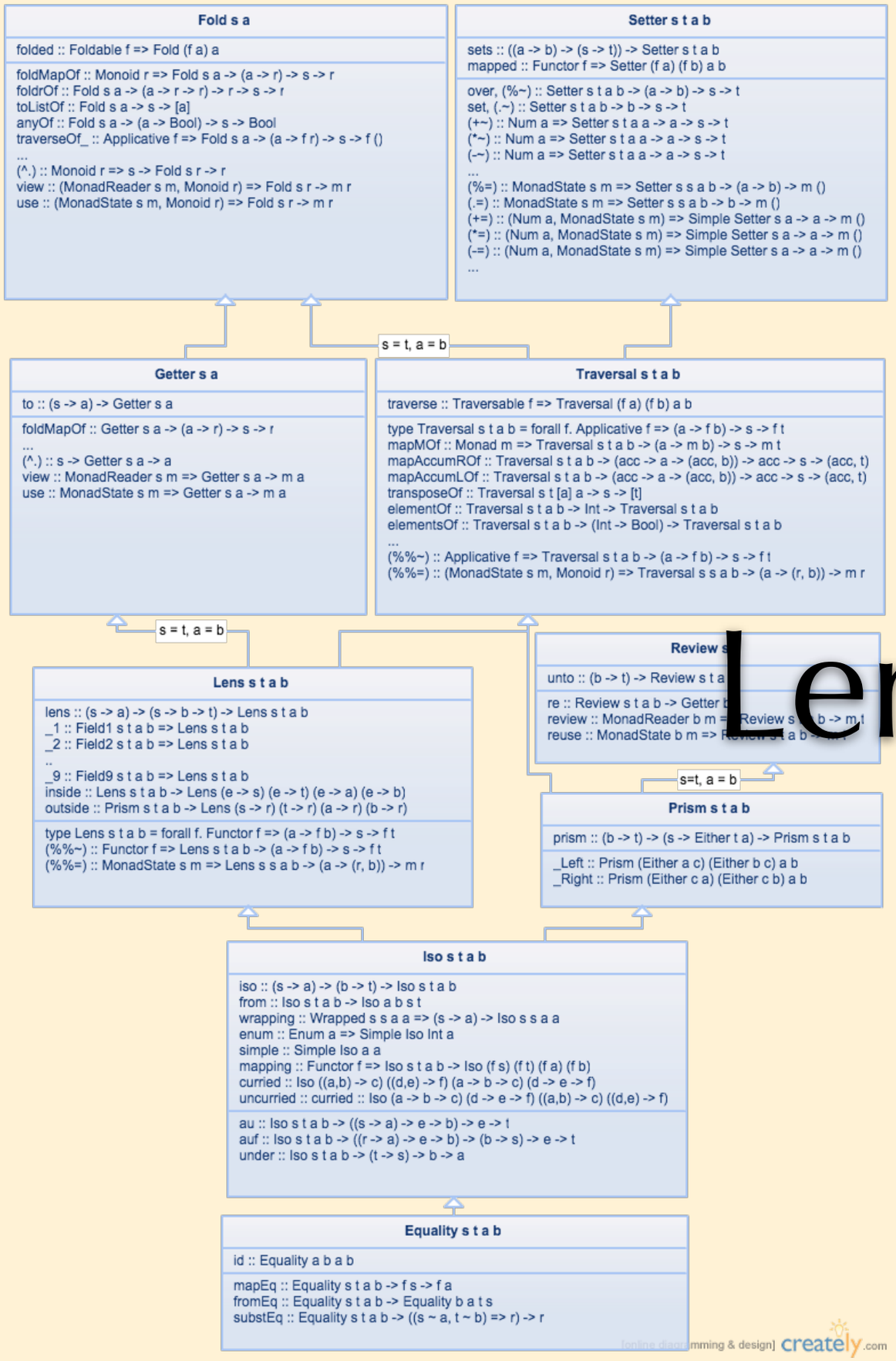
Type Theory	Logic	Category Theory
types	propositions	objects
terms	proofs	arrows
inhabitation of type A	proof of proposition A	arrow $f : T \to A$
function type	implication	exponential
product type	conjunction	product
sum type	disjunction	coproduct
void type (empty type)	falsity	initial object
unit type (singleton type)	truth	terminal object T

WHAT WE HAVE SEEN SO FAR

A Typed λ-Calculus	A Natural Ded. Syst.	A CCC
$\frac{a : A \quad b : B}{\langle a, b\rangle : A \times B}\ (\times_I)$	$\frac{A \quad B}{A \& B}\ (\&_I)$	$\frac{f : T \to A \quad g : T \to B}{\langle f, g\rangle : T \to A \times B}$
$\frac{p : A \times B}{fst\ p : A}\ (\times_{E_1})$	$\frac{A \& B}{A}\ (\&_{E_1})$	$\frac{\langle f, g\rangle : T \to A \times B}{\pi_1 \circ \langle f, g\rangle : T \to A}$
$\frac{p : A \times B}{snd\ p : B}\ (\times_{E_2})$	$\frac{A \& B}{B}\ (\&_{E_2})$	$\frac{\langle f, g\rangle : T \to A \times B}{\pi_2 \circ \langle f, g\rangle : T \to B}$
$\begin{array}{c}[x : A] \\ \vdots \\ b : B\end{array}$	$\begin{array}{c}[A] \\ \vdots \\ B\end{array}$	
$\frac{b : B}{\lambda x.n : A \to B}\ (\to_I)$	$\frac{B}{A \supset B}\ (\supset_I)$	$\frac{g : T \times A \to B}{\Lambda(g) : T \to (A \Rightarrow B)}$
$\frac{f : A \to B \quad x : A}{f\ x : B}\ (\to_E)$	$\frac{A \supset B \quad A}{B}\ (\supset_E)$	$\frac{f : T \to (A \Rightarrow B) \quad g : T \to A}{Ap_{A,B} \circ \langle f, g\rangle : T \to B}$

Mathematics
is
approximation

Lenses



class *Profunctor p* **where**

$$dimap :: (a' \to a) \to (b \to b') \to p\ a\ b \to p\ a'\ b'$$

$$dimap\ id\ id = id$$

$$dimap\ (f' \cdot f)\ (g \cdot g') = dimap\ f\ g \cdot dimap\ f'\ g'$$

instance *Profunctor* (*Lens a b*) **where**

$$dimap\ f\ g\ (Lens\ v\ u) = Lens\ (v \cdot f)\ (g \cdot u \cdot cross\ id\ f)$$

instance *Cartesian* (*Lens a b*) **where**

$$first\ (Lens\ v\ u) = Lens\ (v \cdot fst)\ (fork\ (u \cdot cross\ id\ fst)\ (snd \cdot snd))$$

$$second\ (Lens\ v\ u) = Lens\ (v \cdot snd)\ (fork\ (fst \cdot snd)\ (u \cdot cross\ id\ snd))$$

instance *Profunctor* (*Prism a b*) **where**

$$dimap\ f\ g\ (Prism\ m\ b) = Prism\ (plus\ g\ id \cdot m \cdot f)\ (g \cdot b)$$

instance *Cocartesian* (*Prism a b*) **where**

$$left\ (Prism\ m\ b) = Prism\ (either\ (plus\ Left\ id \cdot m)\ (Left \cdot Right))\ (Left \cdot b)$$

$$right\ (Prism\ m\ b) = Prism\ (either\ (Left \cdot Left)\ (plus\ Right\ id \cdot m))\ (Right \cdot b)$$

Functional languages

- ★ Mathematics-based
- ★ Pure
- ★ Strong sound type system
- ★ Equational reasoning
- ★ $g \circ f$ >> GoF :-)
- ★ No side effects
- ★ Deterministic
- ★ Types as *the* design tool
- ★ Types as *the* source of truth


Real World™

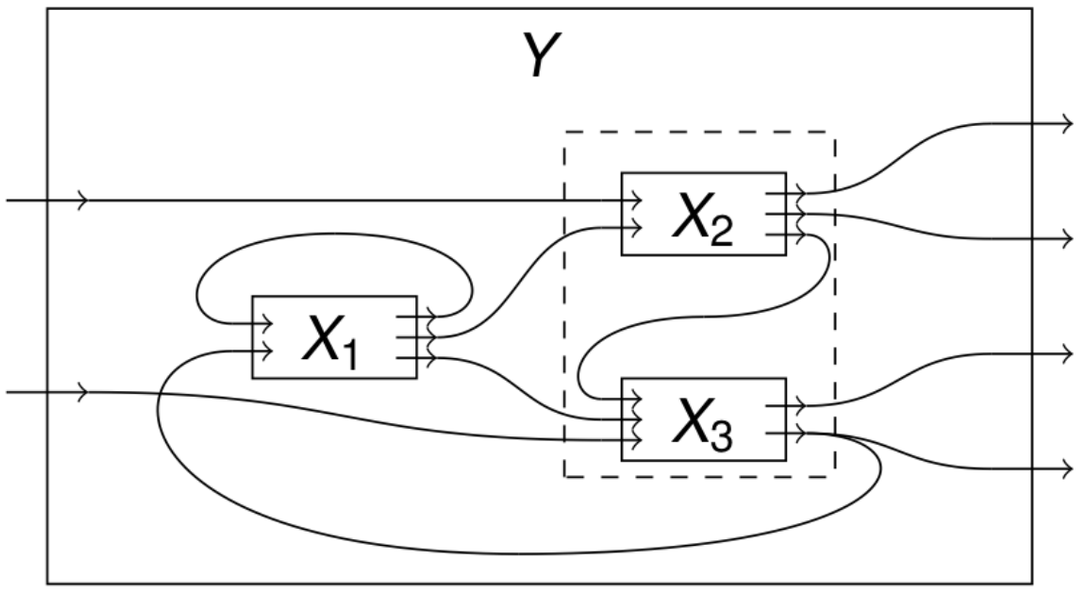
Challenges

EffectsStateMutationsIOAsynchronyConcurrencySecurityIssuesResourceManagementErrorHandling

...

Web applications

- Asynchronous nature
- Concurrency
- State
- Effects
- Interactivity
- Error handling
- Global mutable state (DOM)
- UI representation
- Poor host language



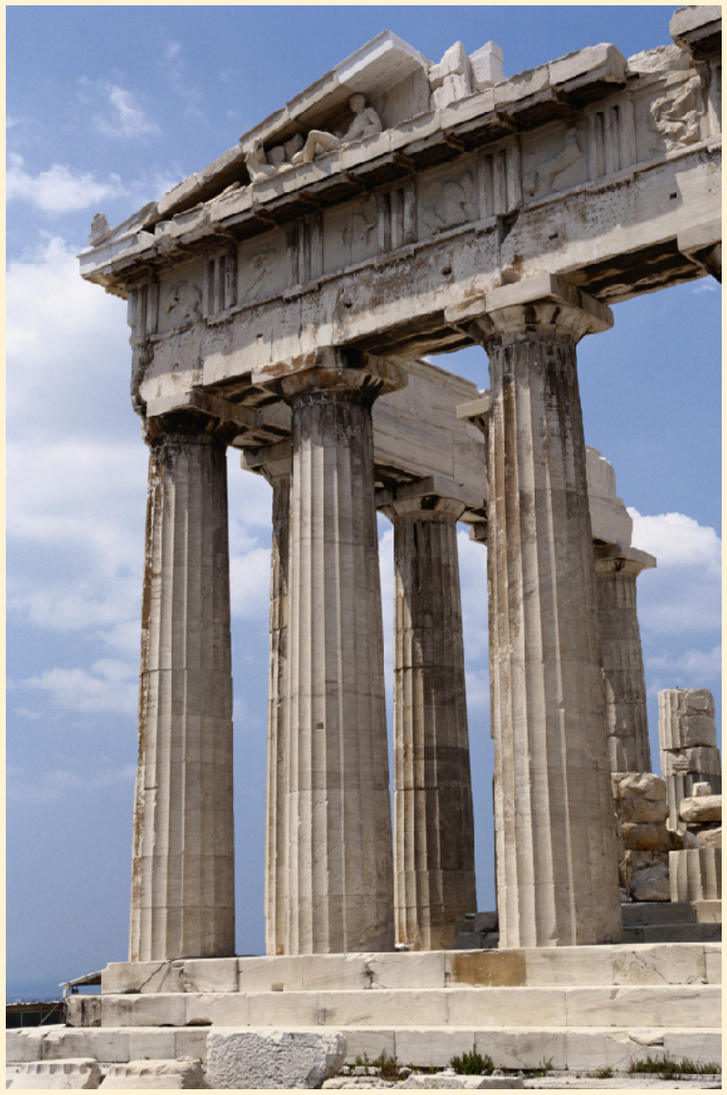
OOP

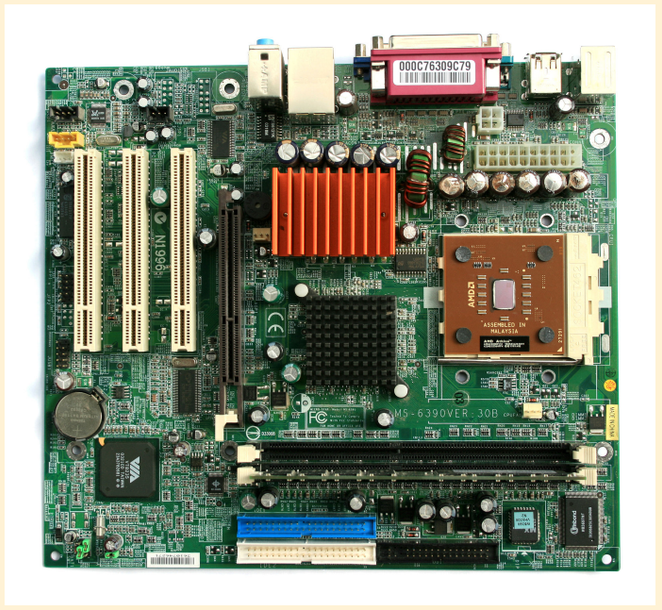
- ★ State encapsulation
- ★ Message passing
- ★ High-level

- ✳ Imperative
- ✳ Mutable
- ✳ Non-deterministic
- ✳ Weak type systems
- ✳ Uncontrolled effects
- ✳ Mixed concerns
- ✳ Concurrency is a mess
- ✳ Ad hoc design

Why state encapsulation?

```
t  ::=
       x
       λx.t
       t t
```





$$a^1 = a \equiv () \to a \equiv a \Leftrightarrow$$ “*there is no essential distinction between having a value and having a (pure) program that computes that value.*”

Software Engineering

Engineering *is the application of knowledge, typically in the form of* ***science****,* ***mathematics****, and empirical evidence, to the innovation, design, construction, operation and maintenance of structures, machines, materials, devices, systems, processes, and organizations.*

State
machines

Synchronous
dataflow

OOP

FRP

Actors

Etc...

MV*

Functional Reactive Animation

Conal Elliott
Microsoft Research
Graphics Group
conal@microsoft.com

Paul Hudak
Yale University
Dept. of Computer Science
paul.hudak@yale.edu

Abstract

Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multimedia animations. The key ideas in Fran are its notions of *behaviors* and *events*. Behaviors are time-varying, reactive values, while events are sets of arbitrarily complex conditions, carrying possibly rich information. Most traditional values can be treated as behaviors, and when images are thus treated, they become animations. Although these notions are captured as data types rather than a programming language, we provide them with a denotational semantics, including a proper treatment of real time, to guide reasoning and implementation. A method to effectively and efficiently perform *event detection* using *interval analysis* is also described, which relies on the partial information structure on the domain of event times. Fran has been implemented in Hugs, yielding surprisingly good performance for an interpreter-based system. Several examples are given, including the ability to describe physical phenomena involving gravity, springs, velocity, acceleration, etc. using ordinary differential equations.

1 Introduction

The construction of richly interactive multimedia animations (involving audio, pictures, video, 2D and 3D graphics) has long been a complex and tedious job. Much of the difficulty, we believe, stems from the lack of sufficiently high-level abstractions, and in particular from the failure to clearly distinguish between *modeling* and *presentation*, or in other words, between *what* an animation is and *how* it should be presented. Consequently, the resulting programs must explicitly manage common implementation chores that have nothing to do with the content of an animation, but rather its presentation through low-level display libraries running on a sequential digital computer. These implementation chores include:

- stepping forward discretely in time for simulation and for frame generation, even though animation is conceptually continuous;
- capturing and handling sequences of motion input events, even though motion input is conceptually continuous;
- time slicing to update each time-varying animation parameter, even though these parameters conceptually vary in parallel; and

By allowing programmers to express the "what" of an interactive animation, one can hope to then automate the "how" of its presentation. With this point of view, it should not be surprising that a set of richly expressive recursive data types, combined with a declarative programming language, serves comfortably for modeling animations, in contrast with the common practice of using imperative languages to program in the conventional hybrid modeling/-presentation style. Moreover, we have found that non-strict semantics, higher-order functions, strong polymorphic typing, and systematic overloading are valuable language properties for supporting modeled animations. For these reasons, Fran provides these data types in the programming language Haskell [9].

Advantages of Modeling over Presentation

The benefits of a modeling approach to animation are similar to those in favor of a functional (or other declarative) programming paradigm, and include clarity, ease of construction, composability, and clean semantics. But in addition there are application-specific advantages that are in some ways more compelling, painting the picture from a software engineering and end-user perspective. These advantages include the following:

- *Authoring.* Content creation systems naturally construct models, because the end users of such systems think in terms of models and typically have neither the expertise nor interest in programming presentation details.
- *Optimizability.* Model-based systems contain a presentation sub-system able to render any model that can be constructed within the system. Because higher-level information is available to the presentation sub-system than with presentation programs, there are many more opportunities for optimization.
- *Regulation.* The presentation sub-system can also more easily determine level-of-detail management, as well as sampling rates required for interactive animations, based on scene complexity, machine speed and load, etc.

To appear the International Conference on Functional Programming, June 1997, Amsterdam.

Functional reactive programming

- Basically *temporal functional programming*

$$\mu :: Behavior\ a \rightarrow (T \rightarrow a)$$

$$\mu :: Event\ a \rightarrow [(T, a)]$$

- Umbrella-term for functional approach to programming reactive systems
- Has evolved in a number of directions and implementations
- Relates to Linear-time temporal logic via Curry–Howard correspondence

Key concepts

- *Behavior a* - a time-varying value
- *Events a* - a time with an associated value
- Functions and instances for *Behaviors* and *Events*
- FRP systems (networks) = function composition
- Locally stateful ~ objects
- Dynamic switching ~ new instance creation
- Deterministic concurrency

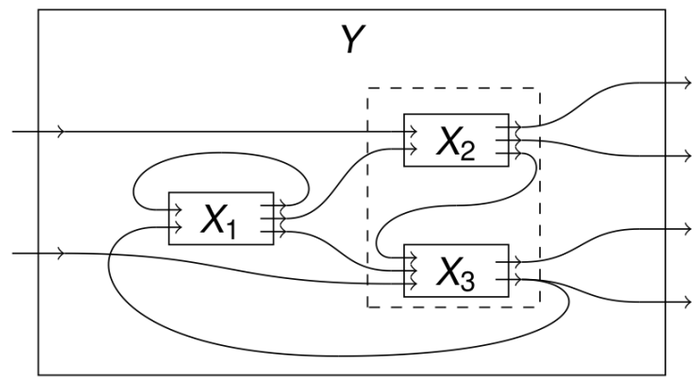
FP addresses:

- ★ Effects
- ★ Error handling
- ★ Global mutable state (DOM)
- ★ UI representation
- ★ Poor host language
- ★ Correctness

×

FRP addresses:

- ★ Asynchronous nature
- ★ Concurrency
- ★ (Local) state
- ★ Interactivity
- ★

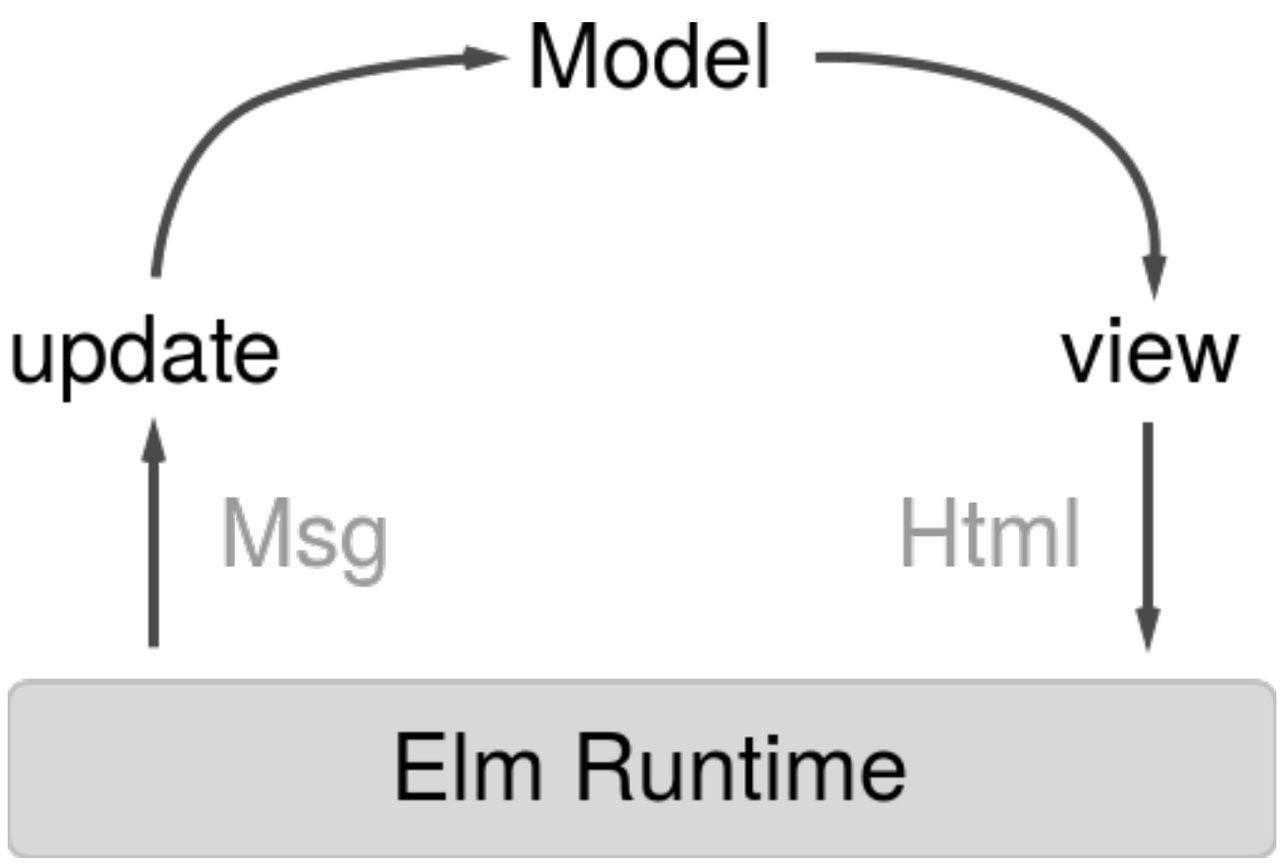


Win × Win

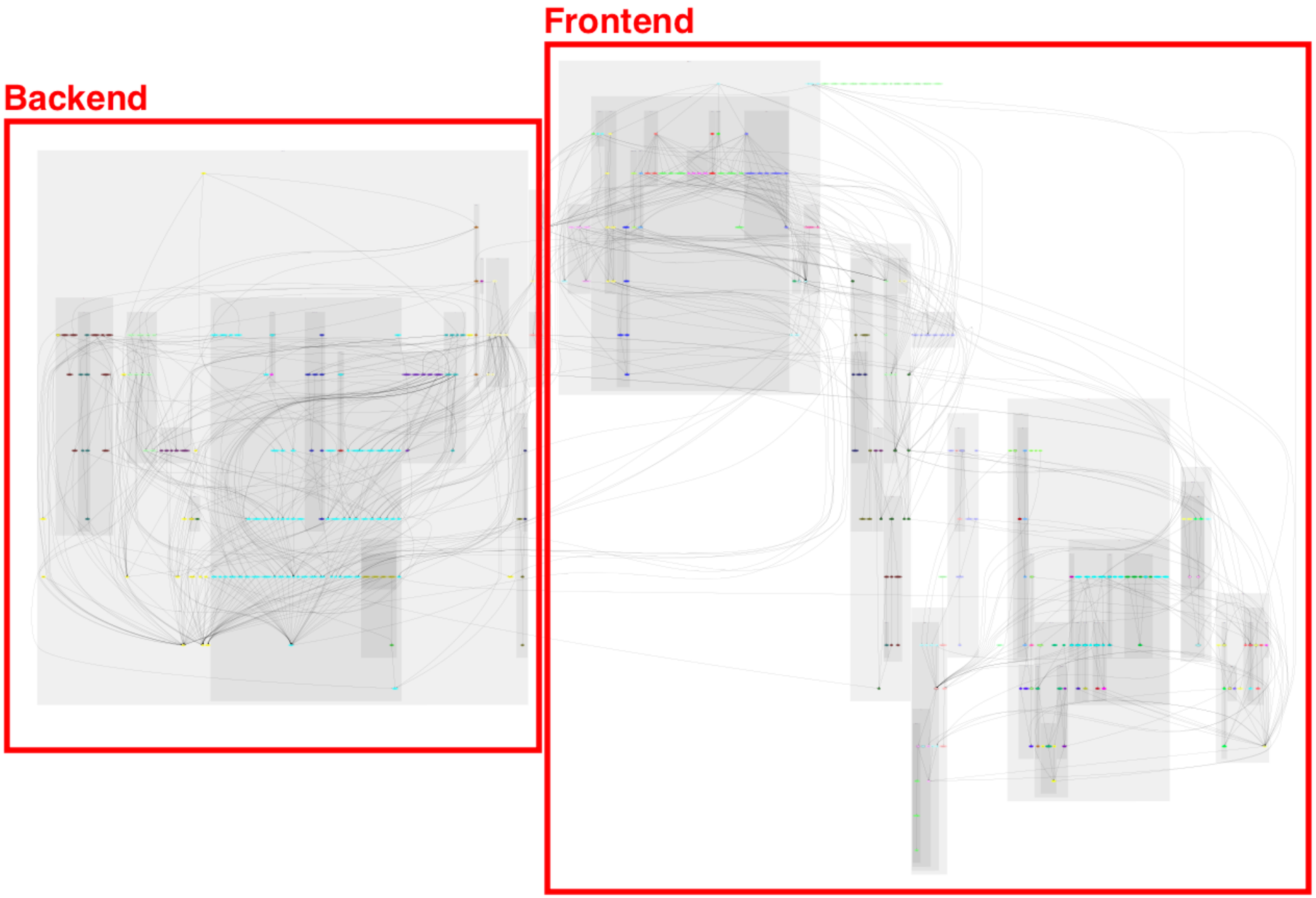
Architecture

Elm vs real

Elm architecture



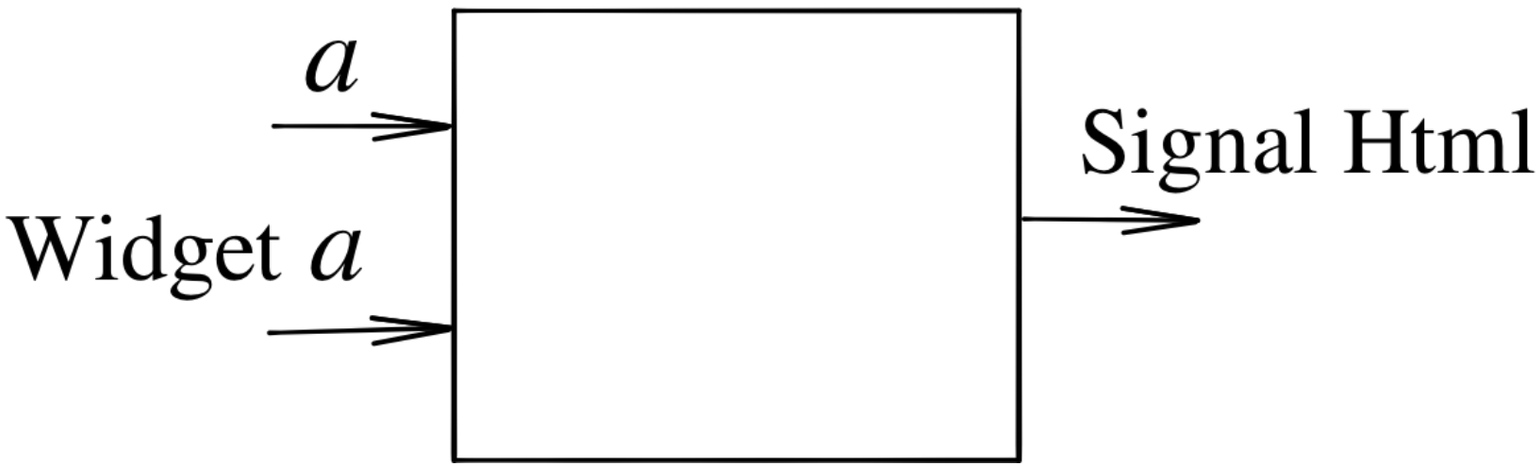
Production architecture



Topologies

UI-only

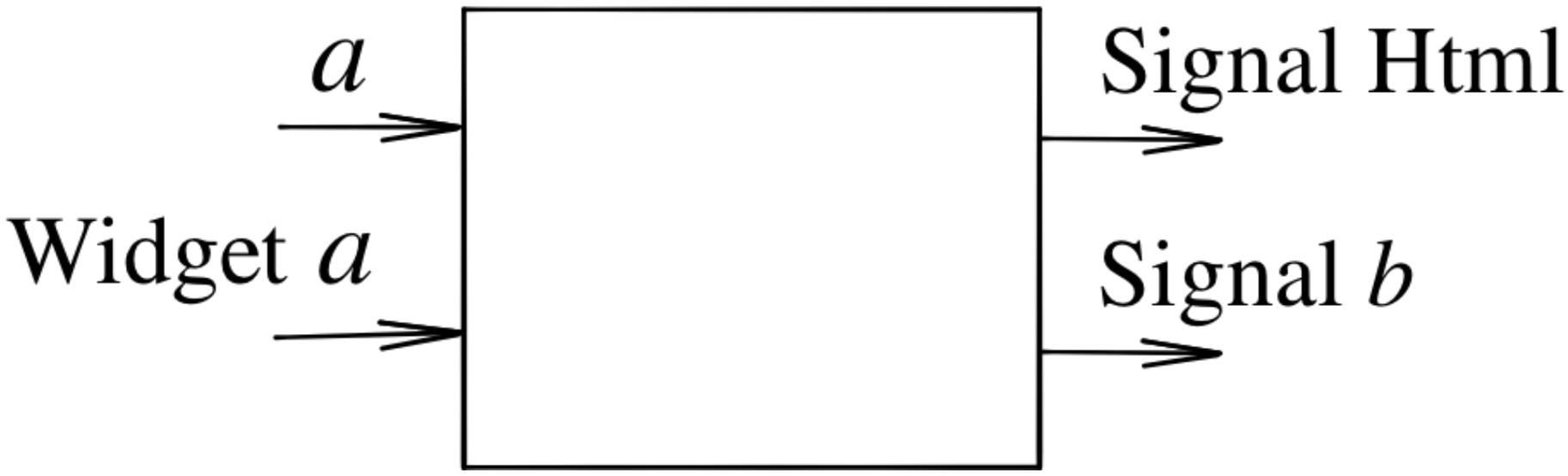
component :: *a* -> Widget *a* -> FRP (Signal HTML)



Read-only

component :: *a* -> Widget *a*

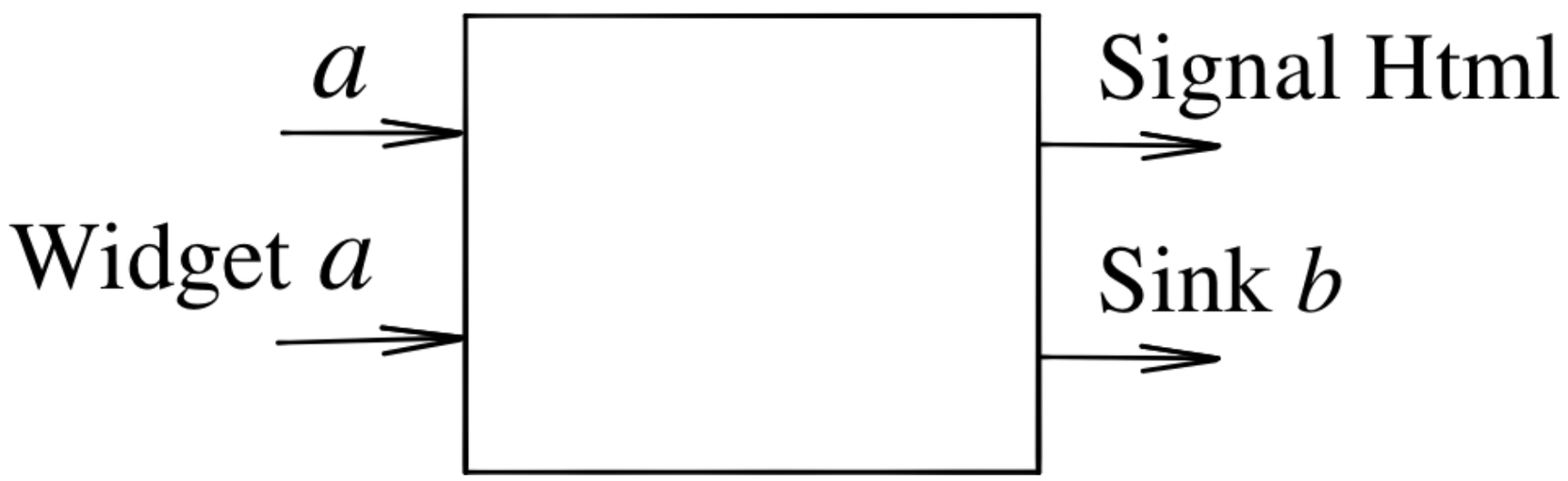
-> FRP (Signal HTML, Signal *b*)



Write-only

component :: *a* -> Widget *a*

-> FRP (Signal HTML, Sink *b*)



Read-Write

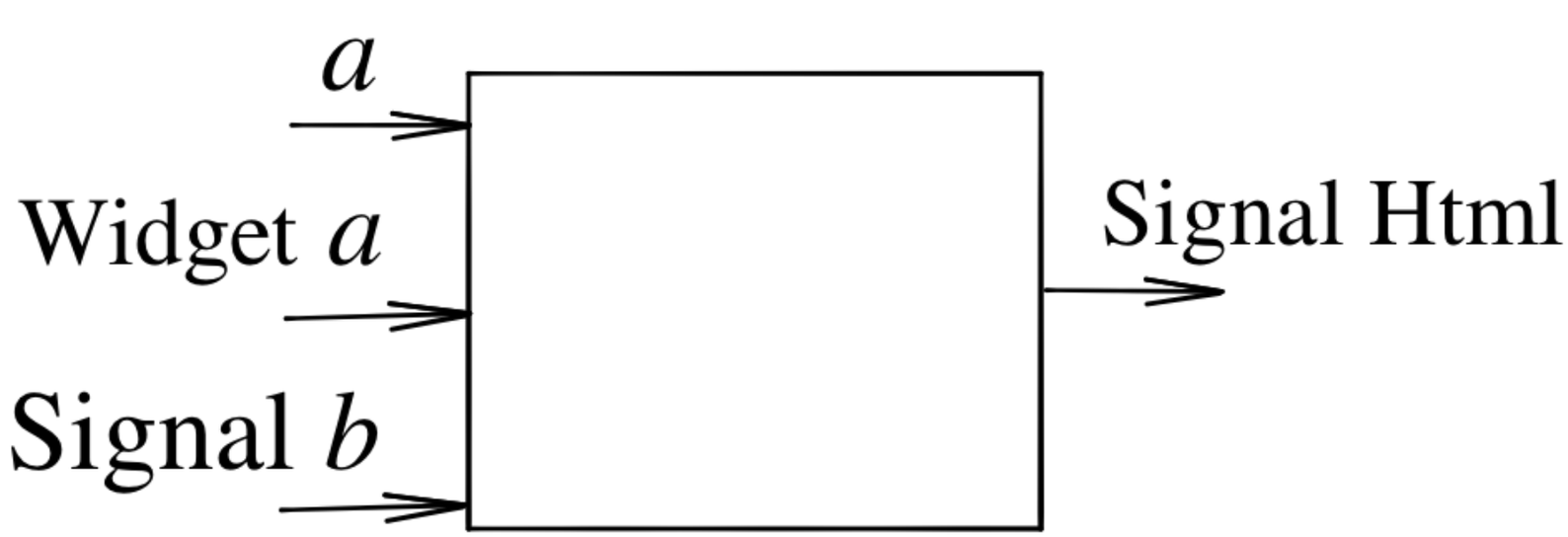
component :: *a* -> Widget *a*

-> FRP (Signal HTML, Sink *b*, Signal *c*)

External read

component :: *a* -> Widget *a* -> Signal *b*

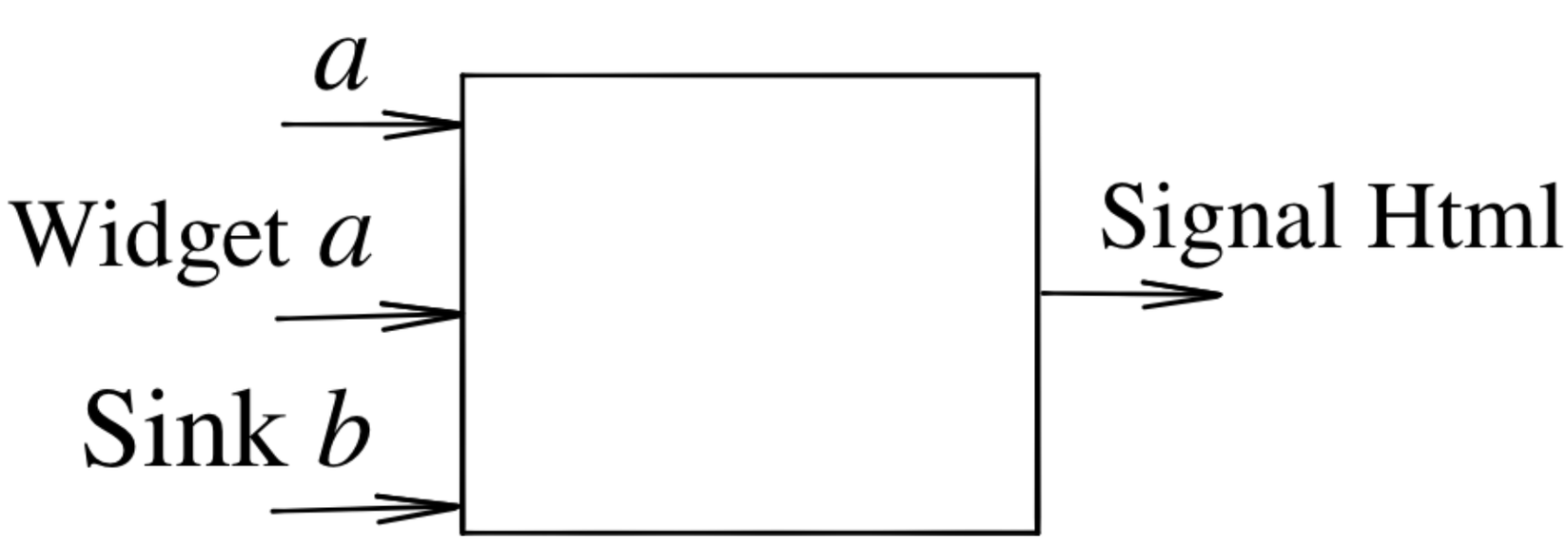
-> FRP (Signal HTML)



External write

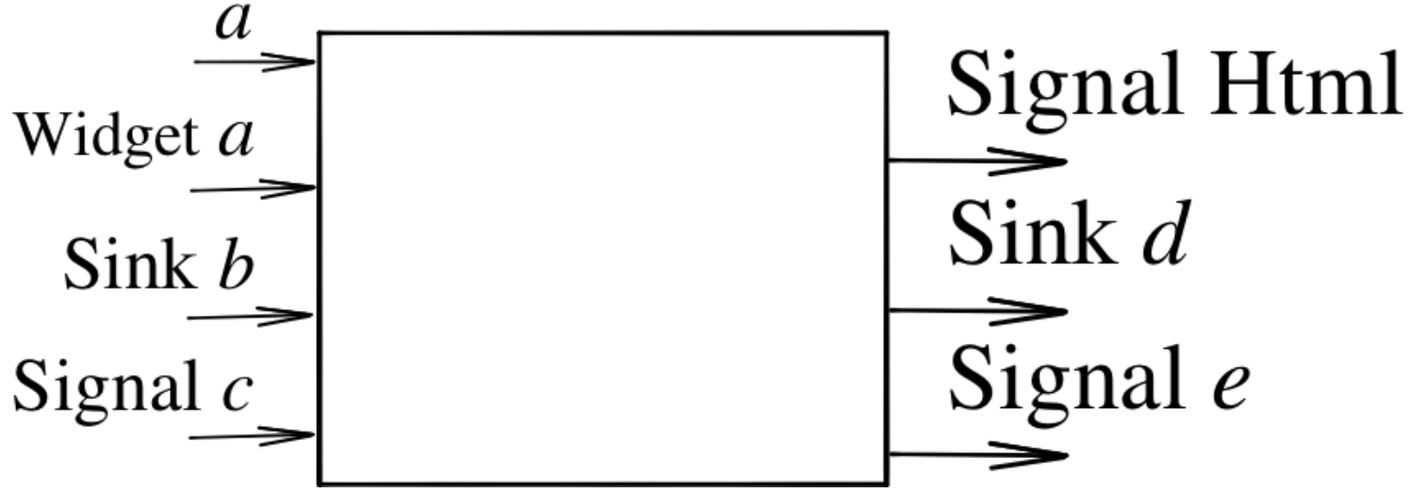
component :: *a* -> Widget *a* -> Sink *b*

-> FRP (Signal HTML)



Uber-component

component :: *a* -> Widget *a* -> Sink *b* -> Signal *c*
-> FRP (Signal HTML, Sink *d*, Signal *e*)



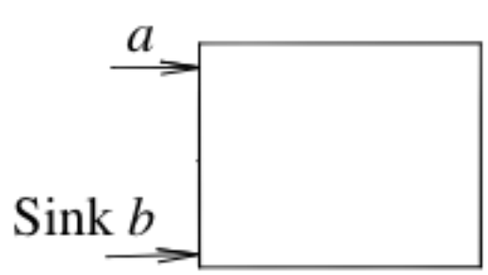
UI-less components

component :: *a* -> FRP (Signal *b*)

component :: *a* -> FRP (Sink *b*, Signal *c*)

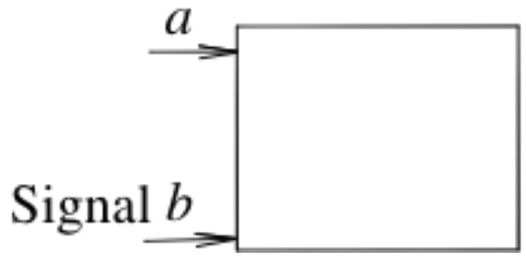
component :: *a* -> FRP (Sink *b*)

component :: *a* -> Sink *b* -> FRP ()

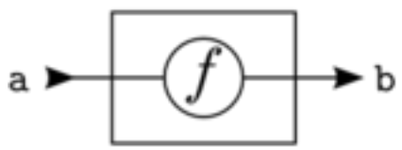


component :: *a* -> Sink *b* -> Signal *c*
-> FRP (Sink *d*, Signal *e*)

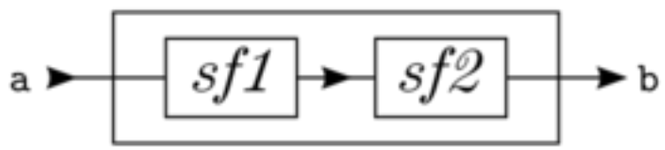
component :: *a* -> Signal *b* -> FRP ()



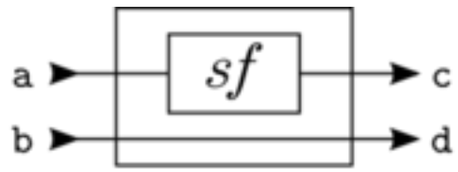
Yampa



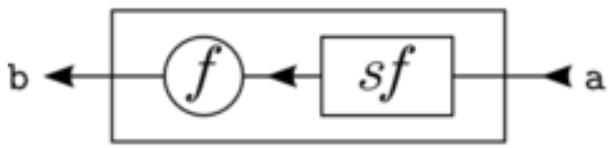
arr f



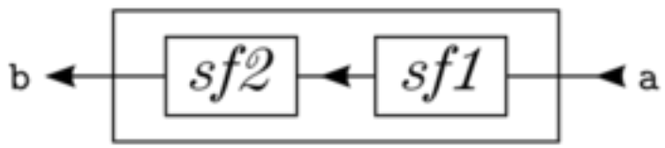
sf1 >>> sf2



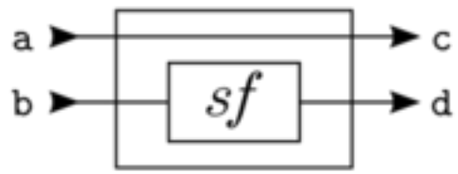
first sf



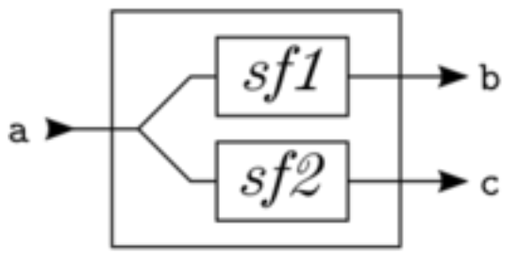
f ^<< sf



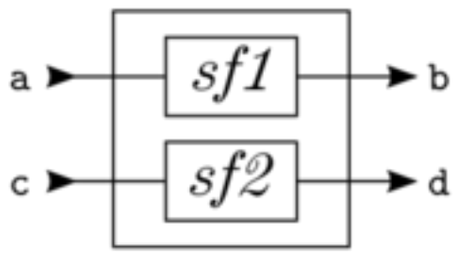
sf2 <<< sf1



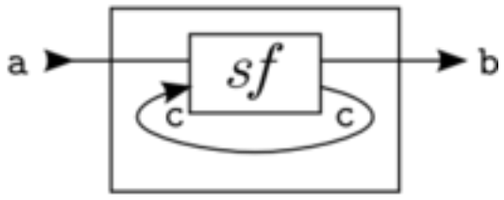
second sf



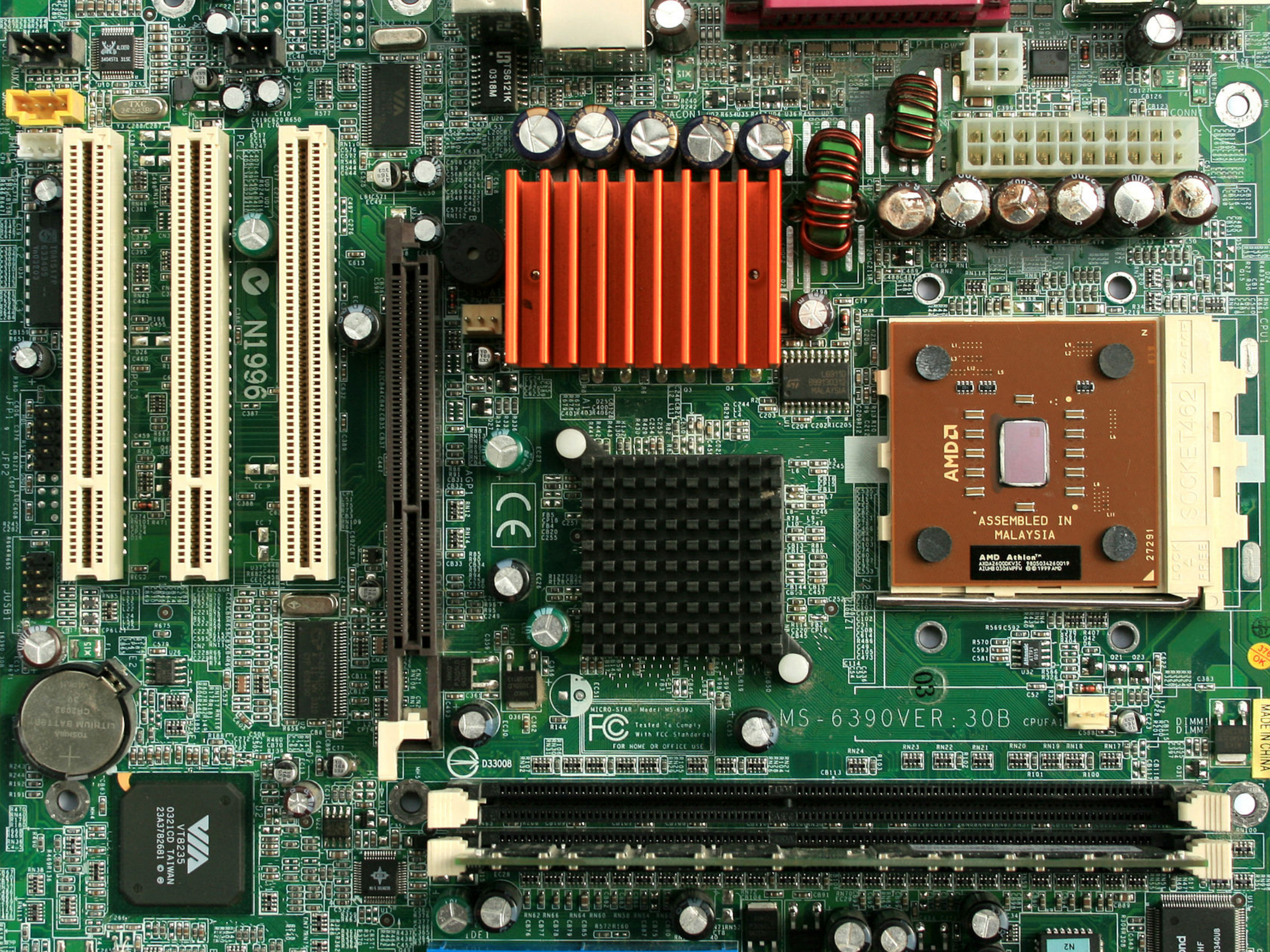
sf1 &&& sf2



sf1 *** sf2



loop sf


MS-6390VER:30B
MICRO-STAR Model MS-6390
Tested To Comply
With FCC Standards
FOR HOME OR OFFICE USE
ASSEMBLED IN
MALAYSIA
AMD Athlon
VT8235
0321CD TAIWAN
MADE IN CHINA

Production experience

- ★ 60kloc in Haskell
- ★ No runtime errors
- ★ No debugging
- ★ No callbacks
- ★ No concurrency issues
- ★ No code duplication
- ★ Safe threads
- ★ Infinite flexibility
- ★ Asynchrony is an implementation detail
- ★ Transfer encoding is an implementation detail
- ★ UI is an implementation detail
- ★ Deterministic sound command-query protocol
- ★ Testing is a joy
- ★ Refactoring is a joy

Demo

(Q→A)→Signal Q→Signal A

Further reading

1. Robert Harper. “Practical Foundations for Programming Languages”
2. Benjamin C. Pierce. “Types and Programming Languages”
3. Blackheath, Stephen; Jones, Antony. ”Functional Reactive Programming”
4. Daniel W. Harris. ”Crash Course in Formal Logic”
5. Steve Awodey. "Category Theory"
6. Bartosz Milewski. "Category Theory for Programmers"
7. Jan-Willem Buurlage. "Categories and Haskell. An introduction to the mathematics behind modern functional programming”
8. Tom Leinster. "Basic Category Theory”

References&copyrights

Slides 2, 42 photos: Copyright (c) Eugene Naumenko
Slide 3, 6: Logos are Copyright (c) respective owners
Slide 4: https://twitter.com/rolyperera/status/1089630313238720513
Slide 6 table: https://www.quora.com/What-are-some-examples-of-functional-programming-languages
Slide 7: https://en.wikipedia.org/wiki/Lambda_cube
Slide 8. https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/krakow.pdf,
http://thinkingwithtypes.com
Slide 9: https://homepages.inf.ed.ac.uk/wadler/papers/propositions-as-types/krakow.pdf,
https://en.wikipedia.org/wiki/Category_theory,
https://homotopytypetheory.org/book/,
https://arxiv.org/abs/1611.02108
https://drive.google.com/file/d/1ghFPbIZ4r8-291f6weszNxLwMXu1U8kx/view
Slide 11: https://github.com/ekmett/lens#lens-lenses-folds-and-traversals,
http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/poptics.pdf
Slide 13: https://www.universetoday.com/41702/picture-of-earth-from-space/
Slide 15, 24: http://math.mit.edu/~dspivak/informatics/talks/CMU2014-01-23.
Slide 17: https://en.m.wikipedia.org/wiki/Motherboard,
http://thinkingwithtypes.com
Slide 19: https://en.wikipedia.org/wiki/Engineering
Slide 21: https://github.com/conal/talk-2015-essence-and-origins-of-frp
http://conal.net/papers/icfp97/
Slide 22: https://www.ioc.ee/~wolfgang/research/mfps-2012-paper.pdf,
https://kodu.ut.ee/~varmo/tday-nelijarve/jeltsch-slides.pdf
Slide 27: https://dennisreimann.de/articles/elm-architecture-overview.html
Slide 28: http://eugenen.github.io/?ui=html#!blog/dd3f0ab84d264b609baead58c817d7dd
Slides 30-37: http://eugenen.github.io/?ui=html#!blog/ddb4287df6585105018e
Slide 38: https://wiki.haskell.org/Yampa
Slide 39: https://en.m.wikipedia.org/wiki/Motherboard